SDP: Final Project  
Nurym Abzal  
SE – 2221

Project overview: Confectionery store

I decided to take a project about a confectionery store for a reason, this is the first clear idea where all the design patterns that I wanted to use in my project would be combined, so I left it that way.

In this project, I used 7 design patterns that provide a plausible service in a pastry shop: Factory, Singleton, Builder, Prototype, Decorator, Observer and Strategy.

the idea of the project is that the client (we) could order various kinds of cakes through a convenient interface. in the course of the order, our order appears on the Order Board. this is done thanks to the Singleton and Observer design patterns.

Here is the implementation:  
ShoppingManager(Singleton):  
  
import java.util.ArrayList;  
import java.util.List;  
  
class ShoppingManager {  
 private static ShoppingManager *instance*;  
 private List<Observer> observers = new ArrayList<>();  
  
 private ShoppingManager() {  
 // private конструктор для Singleton  
 }  
  
 public static synchronized ShoppingManager getInstance() {  
 if (*instance* == null) {  
 *instance* = new ShoppingManager();  
 }  
 return *instance*;  
 }  
  
 public void addObserver(Observer observer) {  
 observers.add(observer);  
 }  
  
 public void removeObserver(Observer observer) {  
 observers.remove(observer);  
 }  
  
 public void addToCart(Cake cake) {  
 // Логика добавления в корзину  
 CakeDecorator cakeDecorator = new CakeDecorator(cake);  
 notifyObservers("Добавлен: " + cakeDecorator.getCake());  
 }  
  
 public void ready(Cake cake) {  
 // Логика оплаты  
 CakeDecorator cakeDecorator = new CakeDecorator(cake);  
 notifyObservers("Оплачен: " + cakeDecorator.getCake());  
 }  
  
 private void notifyObservers(String message) {  
 for (Observer observer : observers) {  
 observer.update(message);  
 }  
 System.*out*.println("");  
 }  
}

in this code we also have Observer, implementation of which is here:

public interface Observer {  
 void update(String message);  
}

public class BoardObserver implements Observer {  
 @Override  
 public void update(String message) {  
 System.*out*.println("В табло заказов " + message);  
 }  
}

public class CardObserver implements Observer {  
 @Override  
 public void update(String message) {  
 System.*out*.println("В чек " + message);  
 }  
}

public class PayCheckObserver implements Observer {  
 @Override  
 public void update(String message) {  
 System.*out*.println(message + " и выдан чек");  
 }  
}

public class RemoveFromBoardObserver implements Observer {  
 @Override  
 public void update(String message) {  
 System.*out*.println(message + " и убран с табло заказов.");  
 }  
}

the main object of sale is a cake around which there are already 4 design patterns(Factory, Builder, Prototype and Decorator):

public interface CakeFactory {  
 Cake createCake();  
}

public class Cake implements Cloneable{  
 private String type;  
 private String size;  
 private String cream;  
 private String filling;  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public String getSize() {  
 return size;  
 }  
  
 public void setSize(String size) {  
 this.size = size;  
 }  
  
 public String getCream() {  
 return cream;  
 }  
  
 public void setCream(String cream) {  
 this.cream = cream;  
 }  
  
 public String getFilling() {  
 return filling;  
 }  
  
 public void setFilling(String filling) {  
 this.filling = filling;  
 }  
  
 @Override  
 public Cake clone() throws CloneNotSupportedException{  
 return (Cake) super.clone();  
 }  
}

in Cake class we used clone method which is implementation of Prototype pattern and following class will “decorate it” by using Decorator pattern, in other words it adds 1 method to display the characteristics of the cake:

public class CakeDecorator {  
 private Cake cake;  
  
 public CakeDecorator(Cake cake){  
 this.cake = cake;  
 }  
  
 public String getCake(){  
 return (this.cake.getType() + " cake with " + this.cake.getSize() + " size, " + this.cake.getCream() +", and " + this.cake.getFilling());  
 }  
}

next come Factory and Builder Pattern, which implement flexibility in the purchase of goods, factory organizes the existence of ready-made cakes when builder allows you to create your own unique cake, allowing you to choose the characteristics of the cake yourself through the console:

public interface CakeFactory {  
 Cake createCake();  
}

public class ChocolateCakeFactory implements CakeFactory {  
 public Cake createCake() {  
 Cake cake = new Cake();  
 cake.setType("Chocolate");  
 cake.setSize("Medium");  
 cake.setCream("Chocolate Cream");  
 cake.setFilling("Chocolate Filling");  
 return cake;  
 }  
}

public class StrawberryCakeFactory implements CakeFactory {  
 public Cake createCake() {  
 Cake cake = new Cake();  
 cake.setType("Strawberry");  
 cake.setSize("Large");  
 cake.setCream("Strawberry Cream");  
 cake.setFilling("Strawberry Filling");  
 return cake;  
 }  
}

public class VanillaCakeFactory implements CakeFactory {  
 public Cake createCake() {  
 Cake cake = new Cake();  
 cake.setType("Vanilla");  
 cake.setSize("Small");  
 cake.setCream("Vanilla Cream");  
 cake.setFilling("Vanilla Filling");  
 return cake;  
 }  
}

here is 3 types of cakes created by Factory design pattern.  
  
Builder:

public class CakeBuilder {  
 private Cake cake;  
  
 public CakeBuilder(String type, String size) {  
 cake = new Cake();  
 cake.setType(type);  
 cake.setSize(size);  
 }  
  
 public CakeBuilder withCream(String cream) {  
 cake.setCream(cream);  
 return this;  
 }  
  
 public CakeBuilder withFilling(String filling) {  
 cake.setFilling(filling);  
 return this;  
 }  
  
 public Cake build() {  
 return cake;  
 }  
}

and last pattern is Strategy which is responsible for the payment strategy:

public interface PaymentStrategy {  
 void pay();  
}

public class CardPaymentStrategy implements PaymentStrategy {  
 @Override  
 public void pay() {  
 System.*out*.println("Оплата картой");  
 }  
}

public class CashPaymentStrategy implements PaymentStrategy {  
 @Override  
 public void pay() {  
 System.*out*.println("Оплата наличными");  
 }  
}

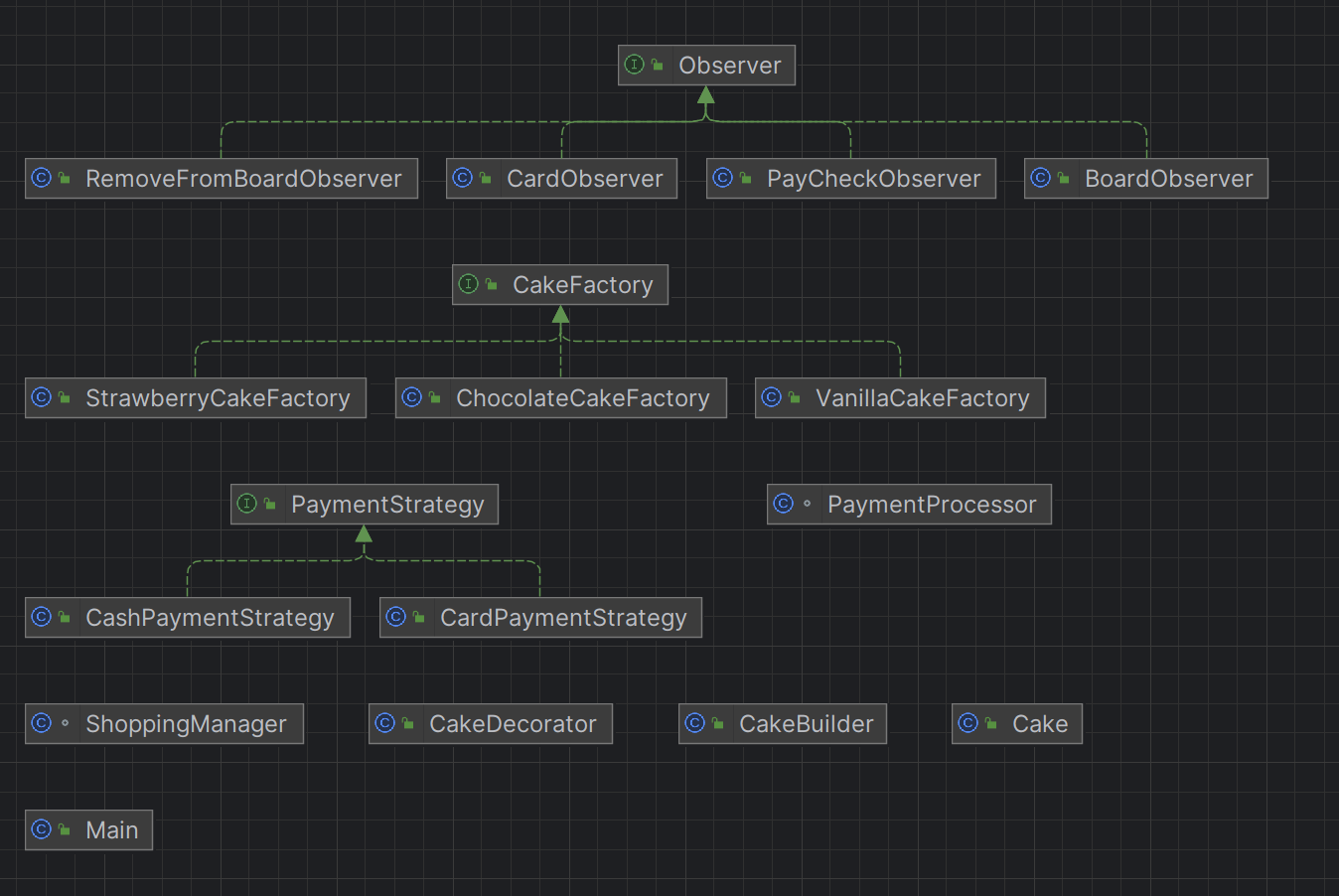
class PaymentProcessor {  
 public void processPayment(PaymentStrategy paymentStrategy) {  
 paymentStrategy.pay();  
 }  
}

Payment Processor is the head of Strategy pattern by which we can choose the payment method at the running time.

And last class is Main class in which we have combination of the all ideas in 1 console menu:

import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 ShoppingManager shoppingManager = ShoppingManager.*getInstance*();  
  
 Observer BoardObserver = new BoardObserver();  
 Observer CardObserver = new CardObserver();  
 Observer removeFromBoardObserver = new RemoveFromBoardObserver();  
 Observer payCheckObserver = new PayCheckObserver();  
  
 shoppingManager.addObserver(BoardObserver);  
 shoppingManager.addObserver(CardObserver);  
  
 /\*<----------------------------factory cakes---------------------------->\*/  
 CakeFactory chocolateCakeFactory = new ChocolateCakeFactory();  
 Cake chocolateCake = chocolateCakeFactory.createCake();  
 CakeDecorator chocolateCakeDecorator = new CakeDecorator(chocolateCake);  
  
 CakeFactory strawberryCakeFactory = new StrawberryCakeFactory();  
 Cake strawberryCake = strawberryCakeFactory.createCake();  
 CakeDecorator strawberryCakeDecorator = new CakeDecorator(strawberryCake);  
 /\*<----------------------------factory cakes---------------------------->\*/  
  
 /\*<----------------------------Prototype---------------------------->\*/  
 Cake clonedCake = new Cake();  
 /\*<----------------------------Prototype---------------------------->\*/  
  
 /\*<----------------------------Pay Cake---------------------------->\*/  
 Cake payCake = new Cake();  
 /\*<----------------------------Pay Cake---------------------------->\*/  
   
 /\*<--------------------------------Menu--------------------------------->\*/  
 System.*out*.println("Добро пожаловать в наш кондитерский магазин!\n Что желаете?\n1 - Выбрать готовый торт.\n2 - Создать свой торт");  
 Scanner sc = new Scanner(System.*in*);  
 int x = sc.nextInt();  
  
 if(x == 1){  
 System.*out*.println("\n\n\nКакой торт выберете?\n1 - " + chocolateCakeDecorator.getCake() + "\n2 - " + strawberryCakeDecorator.getCake());  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 System.*out*.println("Отличный выбор!");  
 shoppingManager.addToCart(chocolateCake);  
 payCake = chocolateCake;  
 }  
 if(x == 2){  
 System.*out*.println("Отличный выбор!");  
 shoppingManager.addToCart(strawberryCake);  
 payCake = strawberryCake;  
 }  
 }  
 if(x == 2){  
 System.*out*.println("Хотите создать новый торт или изменить существующий?\n1 - Создать с нуля\n2 - Изменить существующий");  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 System.*out*.println("Хорошо, введите характеристики торта через ', '!");  
 sc.nextLine();  
 String[] cakeParameters = sc.nextLine().split(", ");  
 Cake myOwnCake = new CakeBuilder(cakeParameters[0], cakeParameters[1]).withCream(cakeParameters[2]).withFilling(cakeParameters[3]).build();  
 System.*out*.println("Отличный выбор!");  
 shoppingManager.addToCart(myOwnCake);  
 payCake = myOwnCake;  
 }  
 if(x == 2){  
 System.*out*.println("Какой торт вы хотите изменить?\n1 - " + chocolateCakeDecorator.getCake() + "\n2 - " + strawberryCakeDecorator.getCake());  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 System.*out*.println("Что хотите изменить в этом торте?\n1 - Начинку\n2 - Крем");  
 sc.nextLine();  
 x = sc.nextInt();  
 clonedCake = chocolateCake.clone();  
 if(x == 1){  
 System.*out*.println("На какую начинку хотите заменить?\n1 - Chocolate Filling\n2 - Strawberry Filling\n3 - Vanilla Filling");  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 clonedCake.setFilling("Chocolate Filling");  
 }  
 if(x == 2){  
 clonedCake.setFilling("Strawberry Filling");  
 }  
 if(x == 3){  
 clonedCake.setFilling("Vanilla Filling");  
 }  
 }  
 else{  
 System.*out*.println("На какой крем хотите заменить?\n1 - Chocolate Cream\n2 - Strawberry Cream\n3 - Vanilla Cream");  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 clonedCake.setCream("Chocolate Cream");  
 }  
 if(x == 2){  
 clonedCake.setCream("Strawberry Cream");  
 }  
 if(x == 3){  
 clonedCake.setCream("Vanilla Cream");  
 }  
 }  
 }  
 else if(x == 2){  
 System.*out*.println("Что хотите изменить в этом торте?\n1 - Начинку\n2 - Крем");  
 sc.nextLine();  
 x = sc.nextInt();  
 clonedCake = strawberryCake.clone();  
 if(x == 1){  
 System.*out*.println("На какую начинку хотите заменить?\n1 - Chocolate Filling\n2 - Strawberry Filling\n3 - Vanilla Filling");  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 clonedCake.setFilling("Chocolate Filling");  
 }  
 if(x == 2){  
 clonedCake.setFilling("Strawberry Filling");  
 }  
 if(x == 3){  
 clonedCake.setFilling("Vanilla Filling");  
 }  
 }  
 else{  
 System.*out*.println("На какой крем хотите заменить?\n1 - Chocolate Cream\n2 - Strawberry Cream\n3 - Vanilla Cream");  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 clonedCake.setCream("Chocolate Cream");  
 }  
 if(x == 2){  
 clonedCake.setCream("Strawberry Cream");  
 }  
 if(x == 3){  
 clonedCake.setCream("Vanilla Cream");  
 }  
 }  
 }  
 System.*out*.println("Отличный выбор!");  
 shoppingManager.addToCart(clonedCake);  
 payCake = clonedCake;  
 }  
 }  
  
 shoppingManager.removeObserver(BoardObserver);  
 shoppingManager.removeObserver(CardObserver);  
  
 shoppingManager.addObserver(removeFromBoardObserver);  
 shoppingManager.addObserver(payCheckObserver);  
  
 PaymentProcessor paymentProcessor = new PaymentProcessor();  
 System.*out*.println("Как будете оплавичать?\n1 - Наличными\n2 - Картой");  
 sc.nextLine();  
 x = sc.nextInt();  
 if(x == 1){  
 paymentProcessor.processPayment(new CashPaymentStrategy());  
 }  
 if(x == 2){  
 paymentProcessor.processPayment(new CardPaymentStrategy());  
 }  
  
 shoppingManager.ready(payCake);  
 /\*<--------------------------------Menu--------------------------------->\*/  
  
 }  
  
}

**UML Diagram**

****

**Conclusion**

The initiative centered around the "confectionery store" has effectively demonstrated the application of multiple design patterns to craft a well-organized and flexible software solution tailored for overseeing the operations of a confectionery store. Throughout the developmental stages, we successfully accomplished the all key milestones

**Challenges faced**

The Selection of Design Patterns: Careful consideration was essential in choosing the suitable design patterns for various aspects of the project. Determining where and how to apply these patterns presented a significant challenge.

Complexity in Implementation: The introduction of multiple design patterns within a single project added complexity. Ensuring the smooth interaction of patterns and preventing code bloat posed challenges during implementation.

Designing the User Interface: The creation of a user-friendly interface, be it command-line or GUI, proved challenging. Striking a balance between functionality, ease of use, and aesthetics necessitated thoughtful design decisions.

Testing and Validation: Continuous challenges were encountered in verifying the correctness and robustness of the code, particularly with the incorporation of multiple design patterns. It was crucial to ensure that the software behaved as expected and handled diverse scenarios.

**Future improvements**

Expanding the product catalog: It is possible to enrich the project by considering the possibility of expanding the range of confectionery products beyond traditional sweets. You can introduce additional confectionery, gourmet treats, or even services related to a candy store.

Improved User Interface: It is possible to improve the user interface by creating a more complex graphical user environment to enhance interaction and create a visually appealing experience for customers exploring confectionery joys.

Database integration: It is possible to implement a database system for storing and managing confectionery store inventory and transaction data. This will allow you to analyze historical sales data, effectively manage inventory and increase the scalability of the confectionery store.

User authentication: It is possible to implement user authentication and authorization functions to ensure secure access to the system. This is especially important for candy stores where different users may have different roles and permissions, such as employees and administrators.

Inventory management: It is possible to develop advanced inventory management functions adapted for a confectionery store, including automatic replenishment of popular products, notifications about a shortage of confectionery products and simplified batch updates for effective inventory control.

**Reference list**

Intellij IDEA: JDK 8.